词法分析作业

路已人 2018211125 2018211302

词法分析作业
- 题目
  - 实验内容及要求
  - 实现方法要求
- 设计说明
  - 字符集
  - 运算符
  - 标识符
  - 字符和字符串
  - 数字
  - 注释
- 自动机构建
  - 标识符
  - 运算符
  - 无符号数字
  - 注释
  - 字符、字符串和预编译指令
  - 完整自动机
- C++实现
  - 匹配过程
    - 错误处理
  - 使用
    - 编译
    - 运行
    - 输出
- Lex实现
- 测试报告
  - 测试代码输入
  - C++语言
    - 输出结果
    - 分析说明
  - Lex
    - 输出结果
      - 分析说明
- 总结
- 源代码
  - C++
  - Lex

题目

C语言词法分析程序的设计与实现

实验内容及要求

1. 可以识别出用C语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
2. 可以识别并跳过源程序中的注释。
3. 可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。
4. 检查源程序中存在的词法错误，并报告错误所在的位置。
5. 对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中存在的所有词法错误。

实现方法要求

1. 采用C/C++作为实现语言，手工编写词法分析程序。
2. 编写LEX源程序，利用LEX编译程序自动生成词法分析程序。

设计说明

字符集

1. 大小写字母、阿拉伯数字
2. `!'" # % & () * + , - . / : ; < = > ?[ \ ] ^ _ { | } ~`
3. `、\n、\t`

运算符

- 单字符：`( ) [ ] { } + - * / % < > ! ~ . , ; ^ " '`
- 双字符：`<= >= == != && || ++ --`

标识符

包含`_`和大小写字母和数字，但是不能以数字开头。

标识符中包含了语言中的保留字：`auto` `break` `case` `char` `const` `continue` `default` `do` `double` `else` `enum` `extern` `float` `for` `goto` `if` `int` `long` `register` `return` `short` `signed` `static` `sizeof` `struct` `switch` `typedef` `union` `unsigned` `void` `volatile` `while` `sizeof`，但是在词法分析阶段我没有对这些关键字做特殊处理。

字符和字符串

字符串以`"`为开头和结尾，可以为空，中间不包含换行符和`"`。可以包含`\`字符，这个字符后面的`"`会被转义而不是识别成字符串的结尾。

字符也类似，区别只是字符不能为空。

数字

数字由符号`+`/`-`和无符号数组成。

对于无符号整数，以`0`开头表示八进制，以`0x`开头表示十六进制，以非零数字开头表示十进制。

整数可包含表示类型的后缀`U`,`L`,`UL`,`LL`,`ULL`，表示`unsigned`类型、`long`类型、`unsigned long`类型、`long long`类型和`unsigned long long`类型。

无符号浮点数只支持10进制表示。支持如下几种形式，（`[x]`表示`x`为可选项，`x|y`表示选取`x`或者`y`）：

`[x].y[e[+|-]z][L]`或者`x[.y][e[+|-]z][L]`（`x`,`y`,`z`为无符号整数）

省略整数部分`x`表示整数部分为`0`。

在结尾添加后缀`L`表示为`long double`类型。

注释

- `//`开头的字符串为单行注释，表示当前行在`//`之后的部分都是注释。
- 由`/*`和`*/`包裹的字符串为多行注释。

自动机构建

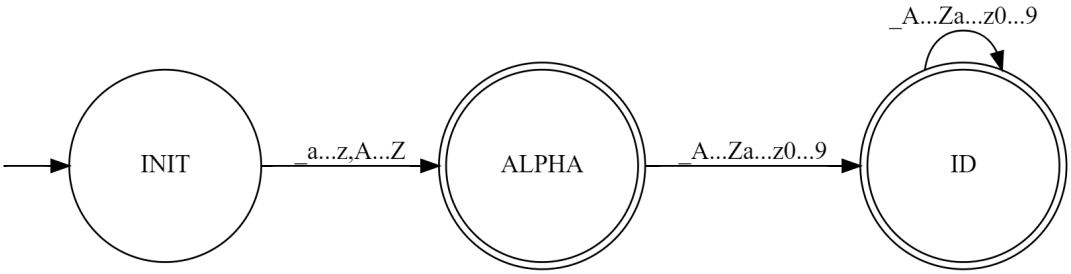
为了实现词法分析的任务，我构造了一个自动机。下面我将逐部分分析这个自动机。

标识符

所有的标识符可由如下文法导出：

$$
\begin{aligned}
q_0 &\rightarrow c_0\,q_1\\
q_1 &\rightarrow c_1\,q_2\\
q_0 &\rightarrow c_1\,q_2\\
c_0 &\rightarrow \_|a|\ldots|z|A|\ldots|Z\\
c_1 &\rightarrow c_0|0|\ldots|9
\end{aligned}
$$

对应到的自动机的部分如图：



从`INIT`到`ALPHA`的一步转移表示接受了一个字母或者`_`，这些字符是可以作为标识符的开始的。接下来出除了这些字符，还可以输入数字来延续这个标识符。

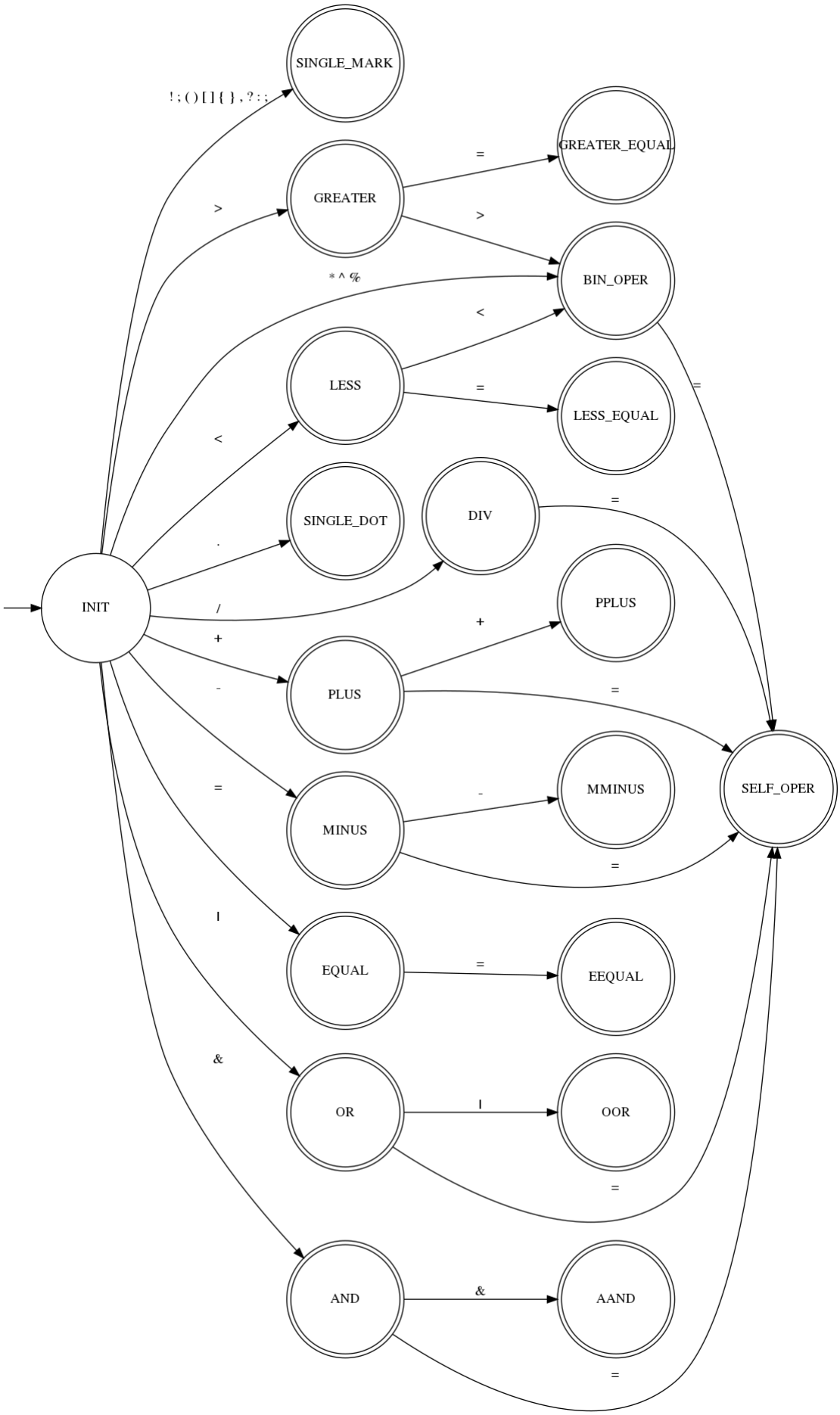
转移到终态的字符串就是一个合法的标识符。

运算符

所有的运算符可由如下文法导出：

$$
\begin{aligned}
q_0 &\rightarrow q_1|q_2|q_3|q_4|q_5|q_6|q_7|q_8|q_9|q_{10}|q_{11} \\
q_1 &\rightarrow !|;|(|)|[|]|\{|\}|,|?|:|; \\
q_2 &\rightarrow < \,|< q_{12} \\
q_{12} &\rightarrow = \,|< q_{13} \\
q_3 &\rightarrow > \,|> q_{14} \\
q_{14} &\rightarrow = \,|> q_{13} \\
q_4 &\rightarrow = \,|= q_{15} \\
q_{15} &\rightarrow = \\
q_5 &\rightarrow +|+q_{16}|+q_{13} \\
q_{16} &\rightarrow + \\
q_6 &\rightarrow -|-q_{17}|-q_{13} \\
q_{17} &\rightarrow - \\
q_7 &\rightarrow \&|\&q_{18}|\&q_{13} \\
q_{18} &\rightarrow \& \\
q_8 &\rightarrow |||q_{19}||q_{13} \\
q_{19} &\rightarrow | \\
q_9 &\rightarrow /|/q_{13} \\
q_{11} &\rightarrow *q_{13}|\wedge q_{13}|\%q_{13} \\
q_{13} &\rightarrow \epsilon|=
\end{aligned}
$$

对应自动机的部分如图

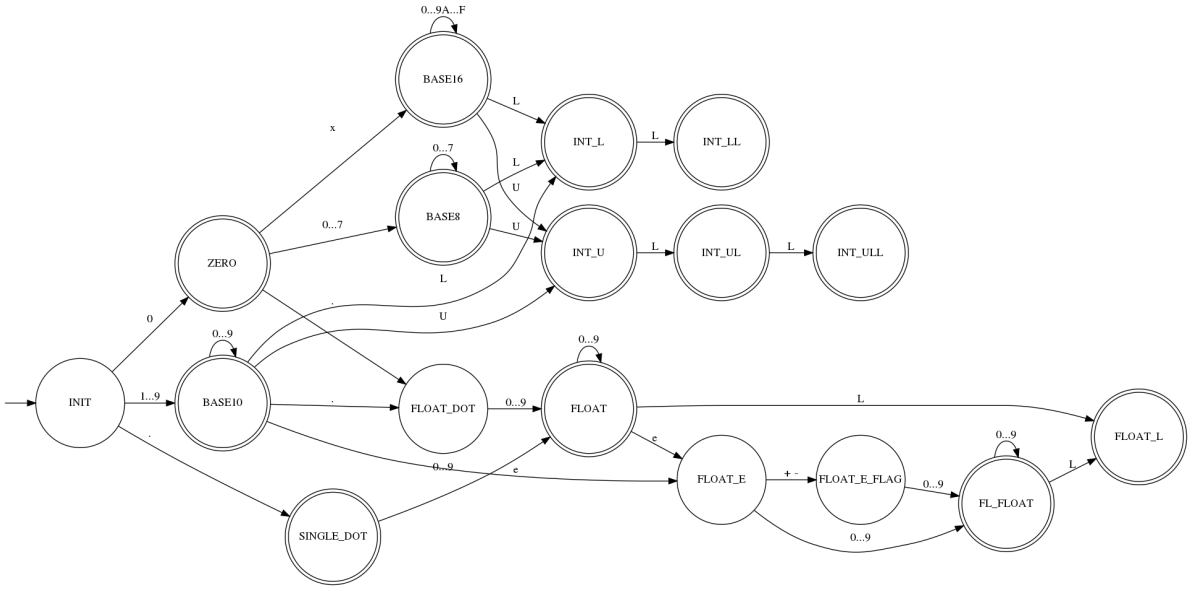


这个运算符的自动机比较麻烦的一点是，由于C语言中存在两个重复的字符组成一个意义不同的运算符的情况，因此，对于这些情况的字符，比如`<`，需要定义一个`LESS`，表示输入一个`<`的状态。从这个状态再输入一个`<`，可以转移到`BIN_OPER`，表示转移到了一个二元运算符——也就是右移`>>`。

无符号数字

数字是所有符号中最复杂的，生成式如下：

$$
\begin{aligned}
q_0 &\rightarrow q_1|q_7 \\
q_1 &\rightarrow 0q_2 \\
q_2 &\rightarrow q_3q_4|xq_5|.q_{10} \\
q_3 &\rightarrow 0|\ldots|7 \\
q_4 &\rightarrow \epsilon|q_3q_4|Uq_{15}|Lq_{16} \\
q_5 &\rightarrow q_6|q_6q_5|Uq_{15}|Lq_{16} \\
q_6 &\rightarrow 0|\ldots|9|A|\ldots|F \\
q_7 &\rightarrow 1q_8|\ldots|9q_8 \\
q_8 &\rightarrow q_9q_8|.q_{10}|eq_{11} \\
q_9 &\rightarrow 0|\ldots|9 \\
q_{10} &\rightarrow q_9q_{12} \\
q_{12} &\rightarrow \epsilon|q_9q_{12}|eq_{11}|L \\
q_{11} &\rightarrow +q_{13}|-q_{13}|q_9q_{14} \\
q_{13} &\rightarrow q_9|q_9q_{14} \\
q_{14} &\rightarrow \epsilon|q_9q_{14}|L \\
q_{15} &\rightarrow \epsilon|Lq_{16} \\
q_{16} &\rightarrow \epsilon|L
\end{aligned}
$$



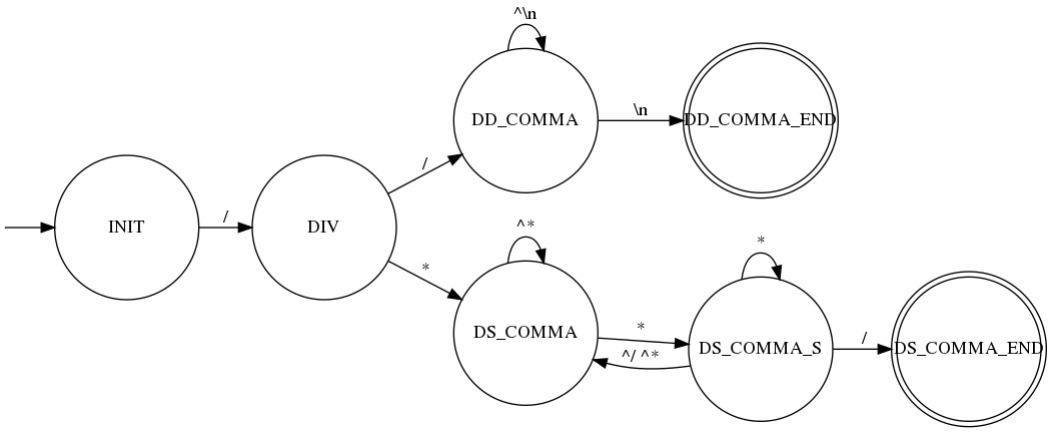
这个自动机比较值得一提的地方是，`.`既可以作为一个浮点数的开头，也可以作为一个单独的运算符，因此需要标记为终态。

注释

注释的格式较为简单，生成式如下：

$$
\begin{aligned}
q_0 &\rightarrow /q_1 \\
q_1 &\rightarrow /q_2|*q_3 \\
q_2 &\rightarrow \text{all}-'\backslash n'q_2|'\backslash n' \\
q_3 &\rightarrow \text{all}-*q_3|*q_4 \\
q_4 &\rightarrow *q_4|\text{all}-*q_3|/
\end{aligned}
$$

对应自动机如图：

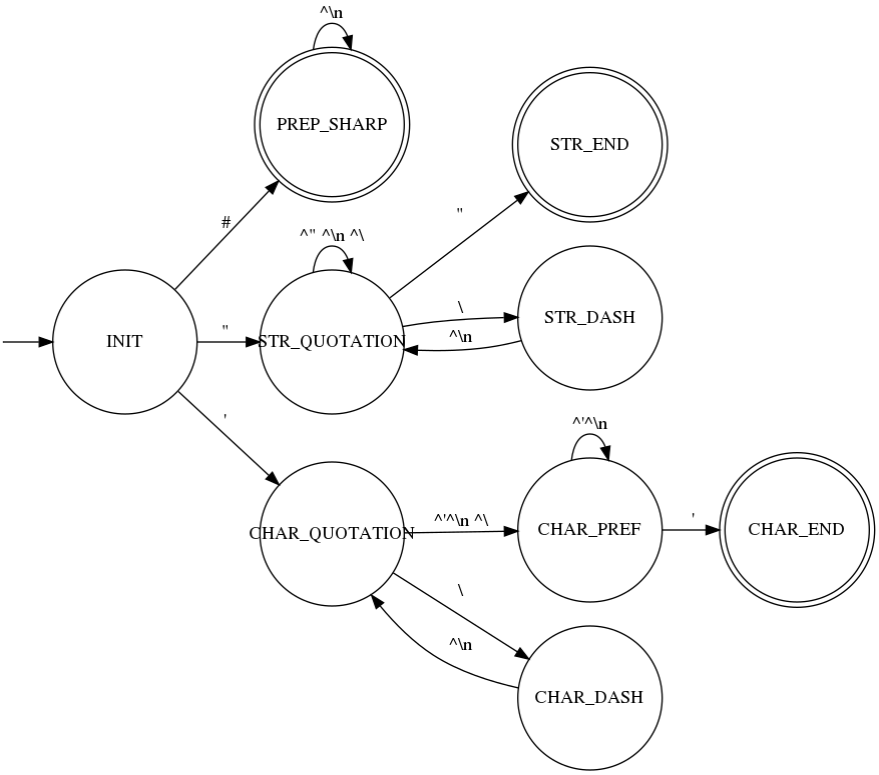


字符、字符串和预编译指令

生成式如下：

$$
\begin{aligned}
q_0 &\rightarrow \#q_1|\text{ ” }q_2|'q_3\\
q_1 &\rightarrow \epsilon|\text{all}-'\ \backslash n'q_1\\
q_2 &\rightarrow \text{all}-\{'\backslash n',\text{”},\backslash\}q_2|\text{ ” }|\backslash q_5\\
q_3 &\rightarrow \text{all}-\{'\backslash n',' ,\backslash\}q_4|\backslash q_6\\
q_4 &\rightarrow \text{all}-\{'\backslash n',' ,\backslash\}q_4|'q_4|\backslash q_6\\
q_5 &\rightarrow \backslash q_3\\
q_6 &\rightarrow \backslash q_4
\end{aligned}
$$

对应自动机如图：



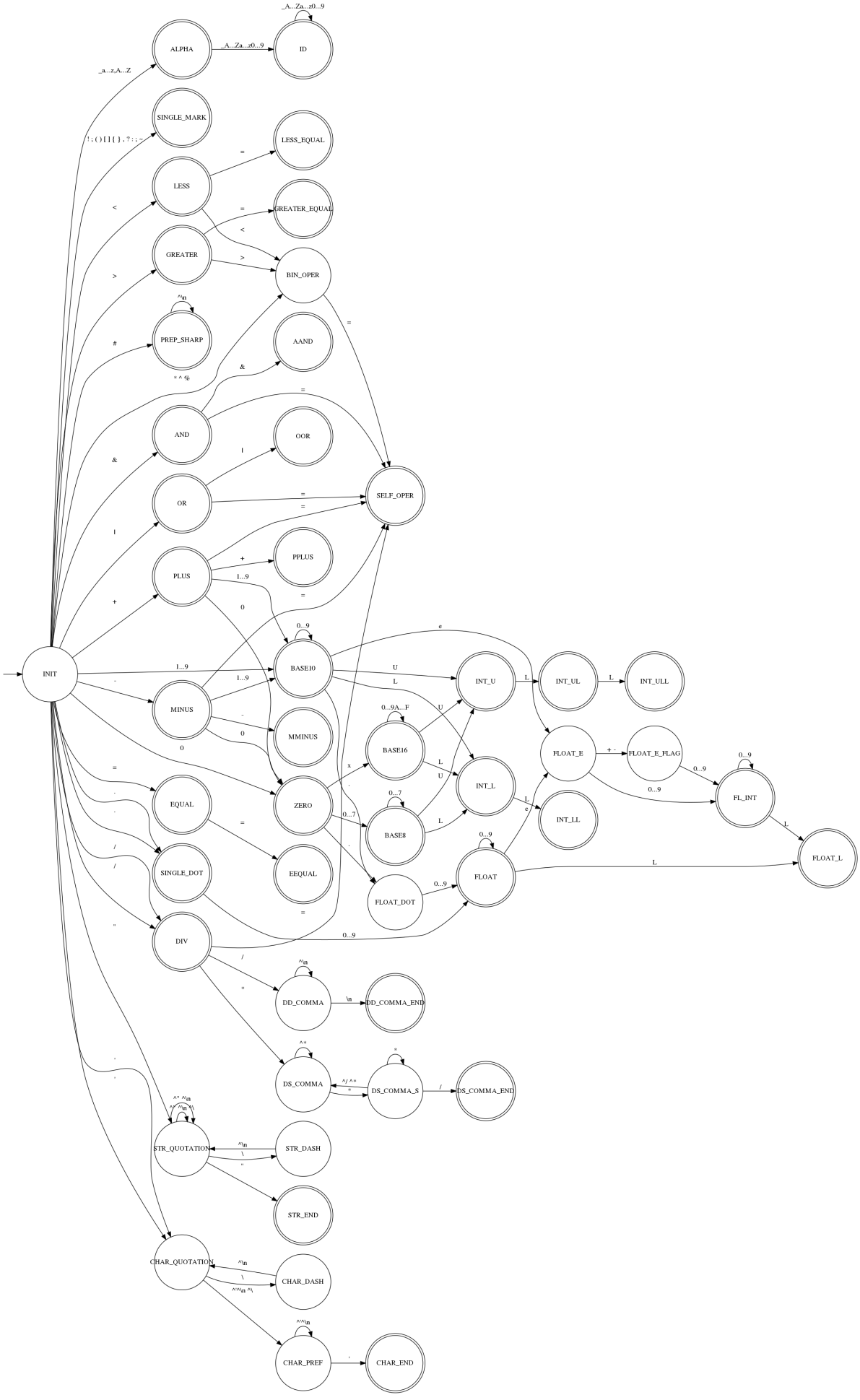
这个自动机中，`STR_DASH`表示，在字符串中输入了一个`\`，那么接下来输入的任意字符都会被转义，包括`"`。输入完成后，再回到`STR_QUOTATION`，此时能够正常接收`"`并终止字符串了。`CHAR_DASH`含义类似。

预编译指令有一个额外的要求，它前面必须是一个换行符，也就是它必须位于行首。

完整自动机

这样，就定义了对于识别每种符号所需要的自动机了。只需它们组合起来编写成程序，就可以完成词法分析的任务了。

完整的自动机如下：



C++实现

使用`enum`将所有的状态和符号类型定义出来，

```
enum STATES {
    FAIL = -1,  // 当前状态没有合适的转移，则转移到失配状态
    INIT,       // 初始状态
    ALPHA,  // 读入了一个字母或者_到达的状态，表示着一个标识符的开始
    SINGLE_MARK,  // 读入一个在词法分析中不起作用的符号到达的状态，如([{等
    DIV,          // 读入一个/
    LESS,         // 读入一个<
    GREATER,      // >
    PREP_SHARP,   // #
    ZERO,         // 0
    BASE10,       //表示十进制数的状态
    SINGLE_DOT,   //读入一个.到达的状态
    AND,          // &
    OR,           // |
    CHAR_QUOTATION,  // '
    CHAR_PREF,       // 读入'和一个字符之后到达的状态
    CHAR_END,        // 表示字符结尾的状态
    PLUS,            // 读入+到达状态
    MINUS,           // -
    EQUAL,           // =
    STR_QUOTATION,   // "
    ID,              // 表示标识符的状态
    DD_COMMA,        // 输入//之后的状态
    DS_COMMA,        // 输入/*之后的状态
    LESS_EQUAL,      // ⩽
    BIN_OPER,        // 表示二元运算符的状态
    GREATER_EQUAL,   // ⩾
    BASE8,           //表示八进制数的状态
    BASE16,          //表示十六进制数字的状态
    FLOAT_DOT,  // 表示输入了浮点数到小数点一位时到达的状态
    AAND,       // &&
    OOR,        // ||
    PPLUS,      // ++
    MMINUS,     // --
    EEQUAL,     // ⩵
    STR_END,    // 表示字符串结尾的状态
    DD_COMMA_END,  // 表示单行注释结束的状态
    DS_COMMA_S,    // 表示多行注释结尾输入了一个*的状态
    SELF_OPER,     // 表示自操作运算符的状态，如+=, ⩾
    INT_U,         //整数结尾输入后缀U的状态
    INT_L,         // 整数结尾输入后缀L的状态
    FLOAT,         // 表示浮点数的状态
    DS_COMMA_END,  //表示多行注释结束的状态
    INT_UL,        //整数结尾输入后缀UL的状态
    INT_LL,        // 整数结尾输入后缀LL的状态
    FLOAT_E,       // 浮点数后输入e的状态
    INT_ULL,       // 整数结尾输入ULL的状态
    FLOAT_E_FLAG,  // 浮点数e后输入了一个+/-号的状态
    FL_INT,        // 浮点数e和+/-号后面跟了一个整数的状态
    FLOAT_L,       // 浮点数后加入L后缀的状态
    STR_DASH,      //正在输入的字符串中输入了一个\到达的状态
    CHAR_DASH      //正在输入的字符中输入了一个\到达的状态
};
```

为了简洁高效地在代码中表示自动机，我设计了如下的类用于表示自动机的状态：

```
using Func = std::function<bool(char)>;

struct State {
  TOKEN_TYPE type;
  std::vector<std::pair<Func, STATES>> go;
  STATES
  Go(char c) {
    for (auto &[pred, s] : go)
      if (pred(c)) return s;
    return FAIL;
  }
};
```

要计算在一个状态加入某个字符会转移到哪个状态，只需要依次遍历所有后继状态，看一下这个字符是否满足转移到某个状态的条件。所有的条件不会有重叠，保证了是一个确定的有限状态自动机。

比如，在识别数字的自动机中，表示已经输入一个浮点数的状态`FLOAT`有如下的转移：

```
{{isdigit, FLOAT},
{Include({'e'}), FLOAT_E},
{Include({'L'}), FLOAT_L}}
```

表示如果来了一个数字，就还是转移到`FLOAT`状态；如果来了一个`e`，就跳转到表示读入了一个浮点数和一个`e`的状态`FLOAT_E`；如果来了一个`L`，就转移到表示浮点数+后缀`L`的状态。

如果没能转移到任何一个状态上，就转移到表示失配的状态`FAIL`上。

匹配过程

在没有读入任何字符之前，在初始态。然后按照读入的字符顺次转移。

如果转移到了一个终态，为了防止这个终态是某个终态的前缀，需要继续处理直到转移到了`FAIL`态，此时再回退一步，此时**期望**会回到一个终态，表示完整的`token`都已经被处理。

对于`1+1`这样的输入，按照优先去匹配更长符号的策略，会被识别成`1`和`+1`。这种情况不方便通过改进自动机解决，但是可以输出到符号列表的时候判断一下，然后将这种相邻数字的后一个数字拆分成运算符+无符号数字的形式。

错误处理

对于词法分析中会遇到的错误，我大致分为了两大类：

1. 匹配过程中没有匹配到一个合法的终状态，比如`0x`，`"aaaaa`，这些串，在自动机上是到不了一个终状态的。
2. 两个符号中间没有合适的分隔被直接连接。比如`1a`这样一个字符串，按照我上面的策略，会先匹配到`1`，尝试下一个字符`a`发现到达了`FAIL`，则回退一步把`1`记录成数字，接下来就会把`a`记录成标识符。但是实际上，数字和标识符是不能紧邻的，虽然他们分别都是合法的符号。

基于上面的分析，我的错误处理策略如下：

1. 对于第一种情况，在自动机中可以解决。如果出现了转移到`FAIL`的情况且回退一步到不了终止状态，就认为出现了错误，此时直接输出已经读入的部分标记为错误，并且回到`INIT`状态。
2. 对于第二种情况，需要在向符号表输出的时候，看一下这个符号能否和符号表结尾的符号直接相邻。比如，数字后面不能紧挨标识符，数字不能紧挨数字，预处理指令前面必须是一个`\n`等等。

经过实践，这样的处理错误策略效果较为不错。

```cpp
std::vector<Token> Run(
    std::ifstream &stream) {  //将流在自动机上匹配输出符号的过程
    std::string token;        //当前输入的符号前缀
    std::vector<Token> tokens;  // 输出的符号列表
    STATES s = INIT;            // 当前状态，初始设置为INIT
    std::vector<int> linec = {
        1};  // 表示每行分别已经输入了多少个字符，用于统计行号和列号
    int line = 1, col = 1;  // 当前符号的开始位置的行列
    bool eof = 0;           // 是否已经读到流的结尾
    for (char c; !eof;) {
        auto Output = [&](TOKEN type, std::string token) {
            tokens.push_back({line, col, type, token});
        };
        if ((c = stream.get()) == EOF) {
            eof = true;
            c = '\n';  // 如果读取到EOF，需要将还没有输出的符号处理完，因此不能直接停止，而是做个标记并且将EOF替换成一个合法的分隔符
        }

        if (!InCharSet(c)) {
            std::cerr << "Unsupported character " << c
                << '\n';  // 不在字符集中的符号，直接忽略
        } else {
            if (c == '\n')  // 新行开始
                linec.push_back(1);
            else
                linec.back()++;       // 当前行多了一个字符
            auto n = state[s].Go(c);  // 计算转移到的状态
            if (s == INIT &&
                n ==
                INIT) {  // 只有空白分隔符能从INIT转移到INIT，此时直接输出不用处理
                Output(TOKEN::BLANK, std::string(1, c));
                line = linec.size();
                col = linec.back();
            } else if (n == FAIL) {  // 失配了
                if (state[s].type ==
                    TOKEN::
                    NOT_END) {  // 上一个状态不是终止态，则认为发生了错误
                    Output(TOKEN::ERR, token += c);  //报错
                    token = "";
                    line = linec.size();
                    col = linec.back();
                } else {
                    stream.unget();  // 回退一个字符
                    if (c == '\n')
                        linec.pop_back();
                    else
                        linec.back()--;
                    Output(state[s].type, token);  // 将已经结束的符号输出
                    token = "";
                    line = linec.size();
                    col = linec.back();  // 新的符号从此处开始
                }
                s = INIT;
            } else {  //没有发生什么问题，平平无奇的转移
                s = n;
                token = token + c;
```

```
            }
        }
    }
    return tokens;
}
```

使用

编译

```
g++ lex.cpp -o lex -std=c++17
```

运行

```
./lex filename
```

输出

输出分为两部分，第一部分是对于不同的符号用不同颜色输出的带行号的代码，错误的符号高亮输出。

第二部分是符号表，具有如下形式

```
line x, column y <TOKEN_TYPE,TOKEN_CONTENT>
```

依次输出了所有非空白符分隔的符号，对于TOKEN_TYPE为ERR的符号，`ERR`被高亮显示。

Lex实现

使用Lex可以根据正则表达式自动生成词法分析程序。我简单学习了一下Lex的基本写法，实现了能够识别我在上文描述的符号的Lex代码。但是，由于Lex是完全的基于正则匹配，我没有想出有效的进行错误的处理的方法。就比如，多行注释只有`/*`没有`*/`，这种情况下，基于正则的匹配会无法匹配到表示注释的字符串，所作的操作就是直接将`/`和`*`按照错误的字符输出，然后继续对剩下的部分按照C语言符号格式来分析，而按照我本人的理解，它应该能够识别出这是一个未闭合的多行注释，同时把`/*`后的部分当作注释字符串直接过掉。我想不出这种方法除了引入类似自动机的思路还能怎么做，因此我没有特别地做错误处理。最后的结果是它对于正确的C代码能够输出的很好，对于有些错误，比如上述的错误，就无法识别到位了。

测试报告

测试代码输入

```
#include <stdio.h>

int main() {
    int a = 123,                    // dec
        b = 0123U,                  // oct unsigned
        c = 0×ABCL,                 // hex long,
        d = 3000000000LL;           // dec long long
    double x = +1e3,                // correct 10^(-3)
        y = .5e-3L,                 // correct 0.5*10^(-3)
        z = +1.2.3.4,               // incorrect
        w = 1A;                     // incorrect
    char str[] = "test\n\\string",  // correct,str with two
```

```
        str_[] = "test;             // incorrect, unmatched quotation
        char ch ='',                // incorrect, empty char
        ch2 = '\n';                 // correct
    for (int i = 1; i <= d || i <= c; ++i) {
        a += i;
        b = a ^ i + 1;
        /* multiple line comment

        // ********************
        */
    }


    /* unclosed multi-line comment
    return 0;
}
```

C++语言

输出结果

Colored Source Code:

```
  1     #include <stdio.h>
  2
  3     int main() {
  4         int a = 123,                    // dec
  5             b = 0123U,                  // oct unsigned
  6             c = 0xABCL,                 // hex long,
  7             d = 3000000000LL;           // dec long long
  8         double x = +1e3,                // correct 10^(-3)
  9             y = .5e-3L,                 // correct 0.5*10^(-3)
 10             z = +1.2.3.4,               // incorrect
 11             w = 1A;                     // incorrect
 12         char str[] = "test\n\\string",   // correct,str with two
 13             str_[] = "test;             // incorrect, unmatched quotation
 14             char ch = '',               // incorrect, empty char
 15             ch2 = '\n';                 // correct
 16         for (int i = 1; i <= d || i <= c; ++i) {
 17             a += i;
 18             b = a ^ i + 1;
 19             /* multiple line comment
 20
 21             // ********************
 22             */
 23         }
 24
 25
 26         /* unclosed multi-line comment
 27         return 0;
 28     }
```

28 lines in total.

Token List:

```
line 3, column 2:     <ID,int>
line 3, column 6:     <ID,main>
line 3, column 10:    <SINGLE_MARK,(>
line 3, column 11:    <SINGLE_MARK,)>
line 3, column 13:    <SINGLE_MARK,{>
line 4, column 6:     <ID,int>
line 4, column 10:    <ID,a>
line 4, column 12:    <ASSIGN,=>
line 4, column 14:    <INT,123>
line 4, column 17:    <SINGLE_MARK,,>
line 5, column 10:    <ID,b>
line 5, column 12:    <ASSIGN,=>
line 5, column 14:    <INT,0123U>
line 5, column 19:    <SINGLE_MARK,,>
line 6, column 10:    <ID,c>
line 6, column 12:    <ASSIGN,=>
line 6, column 14:    <INT,0xABCL>
line 6, column 20:    <SINGLE_MARK,,>
line 7, column 10:    <ID,d>
line 7, column 12:    <ASSIGN,=>
line 7, column 14:    <INT,3000000000LL>
line 7, column 26:    <SINGLE_MARK,;>
line 8, column 6:     <ID,double>
line 8, column 13:    <ID,x>
line 8, column 15:    <ASSIGN,=>
line 8, column 17:    <FLOAT,+1e3>
line 8, column 21:    <SINGLE_MARK,,>
line 9, column 10:    <ID,y>
line 9, column 12:    <ASSIGN,=>
line 9, column 14:    <FLOAT,.5e-3L>
line 9, column 20:    <SINGLE_MARK,,>
line 10, column 10:   <ID,z>
line 10, column 12:   <ASSIGN,=>
line 10, column 14:   <ERR,+1.2.3.4>
line 10, column 22:   <SINGLE_MARK,,>
line 11, column 10:   <ID,w>
line 11, column 12:   <ASSIGN,=>
line 11, column 14:   <ERR,1A>
line 11, column 16:   <SINGLE_MARK,;>
line 12, column 6:    <ID,char>
line 12, column 11:   <ID,str>
line 12, column 14:   <SINGLE_MARK,[>
line 12, column 15:   <SINGLE_MARK,]>
line 12, column 17:   <ASSIGN,=>
line 12, column 19:   <STR,"test\n\\string">
line 12, column 35:   <SINGLE_MARK,,>
line 13, column 10:   <ID,str>
line 13, column 13:   <ID,_>
line 13, column 14:   <SINGLE_MARK,[>
line 13, column 15:   <SINGLE_MARK,]>
line 13, column 17:   <ASSIGN,=>
line 13, column 19:   <ERR,"test;             // incorrect, unmatched quotation\n>
line 14, column 10:   <ID,char>
line 14, column 15:   <ID,ch>
```

```
line 14, column 18:   <ASSIGN,=>
line 14, column 19:   <ERR,''>
line 14, column 21:   <SINGLE_MARK,,>
line 15, column 10:   <ID,ch2>
line 15, column 14:   <ASSIGN,=>
line 15, column 16:   <CHAR,'\n'>
line 15, column 20:   <SINGLE_MARK,;>
line 16, column 6:    <ID,for>
line 16, column 10:   <SINGLE_MARK,(>
line 16, column 11:   <ID,int>
line 16, column 15:   <ID,i>
line 16, column 17:   <ASSIGN,=>
line 16, column 19:   <INT,1>
line 16, column 20:   <SINGLE_MARK,;>
line 16, column 22:   <ID,i>
line 16, column 24:   <RELOP,<=>
line 16, column 27:   <ID,d>
line 16, column 29:   <OPER,||>
line 16, column 32:   <ID,i>
line 16, column 34:   <RELOP,<=>
line 16, column 37:   <ID,c>
line 16, column 38:   <SINGLE_MARK,;>
line 16, column 40:   <OPER,++>
line 16, column 42:   <ID,i>
line 16, column 43:   <SINGLE_MARK,)>
line 16, column 45:   <SINGLE_MARK,{>
line 17, column 10:   <ID,a>
line 17, column 12:   <OPER,+=>
line 17, column 15:   <ID,i>
line 17, column 16:   <SINGLE_MARK,;>
line 18, column 10:   <ID,b>
line 18, column 12:   <ASSIGN,=>
line 18, column 14:   <ID,a>
line 18, column 16:   <OPER,^>
line 18, column 18:   <ID,i>
line 18, column 20:   <OPER,+>
line 18, column 22:   <INT,1>
line 18, column 23:   <SINGLE_MARK,;>
line 23, column 6:    <SINGLE_MARK,}>
line 26, column 6:    <ERR,/* unclosed multi-line comment\n    return 0;\n}>


Token Counts:
Number of          ASSIGN       14
Number of           BLANK       350
Number of            CHAR       1
Number of           COMMA       12
Number of             ERR       5
Number of           FLOAT       2
Number of              ID       32
Number of             INT       6
Number of            OPER       5
Number of            PREP       1
Number of           RELOP       2
Number of     SINGLE_MARK       26
Number of             STR       1


Char Counts: 899

# Lucida /mnt/d/Lex on git:master x [11:12:12]
$
```

分析说明

代码中的词法错误在注释里有指明，可以看出，程序正确指出了代码中的词法错误。

程序中的符号、数字、运算符等词法元素均被正确处理，并且正确地统计了各种类型的符号的个数和字符个数以及行数。

Lex

输出结果

```
$ lex lex.i && gcc lex.yy.c -lfl -o lex-lex && ./lex-lex test.c
<PREP,#include <stdio.h>
>
<ID,int>
<ID,main>
<SINGLE_MARK,(>
<SINGLE_MARK,)>
<SINGLE_MARK,{>
<ID,int>
<ID,a>
<ASSIGN,=>
<INT,123>
<SINGLE_MARK,,>
<COMMA,// dec
>
<ID,b>
<ASSIGN,=>
<INT,0123U>
<SINGLE_MARK,,>
<COMMA,// oct unsigned
>
<ID,c>
<ASSIGN,=>
<INT,0xABCL>
<SINGLE_MARK,,>
<COMMA,// hex long,
>
<ID,d>
<ASSIGN,=>
<INT,3000000000LL>
<SINGLE_MARK,;>
<COMMA,// dec long long

<ID,double>
<ID,x>
<ASSIGN,=>
<FLOAT,+1e3>
<SINGLE_MARK,,>
<COMMA,// correct 10^(-3)
>
<ID,y>
<ASSIGN,=>
<SINGLE_MARK,.>
<FLOAT,5e-3L>
<SINGLE_MARK,,>
<COMMA,// correct 0.5*10^(-3)
>
<ID,z>
<ASSIGN,=>
<FLOAT,+1.2>
<SINGLE_MARK,.>
<FLOAT,3.4>
<SINGLE_MARK,,>
<COMMA,// incorrect
>
<ID,w>
<ASSIGN,=>
<INT,1>
<ID,A>
<SINGLE_MARK,;>
<COMMA,// incorrect
>
<ID,char>
<ID,str>
<SINGLE_MARK,[>
<SINGLE_MARK,]>
<ASSIGN,=>
```

```
<STR,"test\n\\string">
<SINGLE_MARK,,>
<COMMA,// correct,str with two
>
<ID,str_>
<SINGLE_MARK,[>
<SINGLE_MARK,]>
<ASSIGN,=>
<ERR,">
<ID,test>
<SINGLE_MARK,;>
<COMMA,// incorrect, unmatched quotation
>
<ID,char>
<ID,ch>
<ASSIGN,=>
<ERR,'>
<ERR,'>
<SINGLE_MARK,,>
<COMMA,// incorrect, empty char

<ID,ch2>
<ASSIGN,=>
<CHAR,'\n'>
<SINGLE_MARK,;>
<COMMA,// correct
>
<ID,for>
<SINGLE_MARK,(>
<ID,int>
<ID,i>
<ASSIGN,=>
<INT,1>
<SINGLE_MARK,;>
<ID,i>
<RELOP,<=>
<ID,d>
<OPER,|>
<OPER,|>
<ID,i>
<RELOP,<=>
<ID,c>
<SINGLE_MARK,;>
<OPER,+>
<OPER,+>
<ID,i>
<SINGLE_MARK,)>
<SINGLE_MARK,{>
<ID,a>
<OPER,+>
<ASSIGN,=>
<ID,i>
<SINGLE_MARK,;>
<ID,b>
<ASSIGN,=>
<ID,a>
<OPER,^>
<ID,i>
<OPER,+>
<INT,1>
<SINGLE_MARK,;>
<COMMA,/* multiple line comment

        // ********************
        */>
<SINGLE_MARK,}>
<OPER,/>
<OPER,*>
```

```
<ID,unclosed>
<ID,multi>
<OPER,->
<ID,line>
<ID,comment>
<ID,return>
<INT,0>
<SINGLE_MARK,;>
<SINGLE_MARK,}>

# Lucida /mnt/d/Lex on git:master x [16:34:15]
$
```

分析说明

对于合法的符号，程序正确输出了结果。对于非法的符号，程序直接将无法解释的字符直接扔掉，继续分析剩下的部分，我认为这不太合适。但是鉴于这种使用正则的方式代码非常简洁，我也就不要求过高了。

总结

在本次作业中，我对于编译中的词法分析有了更深刻的认识，对于有限状态自动机的构造和应用更加熟练了。

在编程实现的过程中，我遇到了很多问题和错误，其中很多问题使得我对词法错误有了更深入的思考。

在思维方面收获的同时，我学习了Graphviz和Lex工具的简单使用。

虽然我在本次作业中投入很多时间，但是收获也非常大。

源代码

C++

```cpp
#include <bits/stdc++.h>

enum STATES {
    FAIL = -1,  // 当前状态没有合适的转移，则转移到失配状态
    INIT,       // 初始状态
    ALPHA,  // 读入了一个字母或者_到达的状态，表示着一个标识符的开始
    SINGLE_MARK,  // 读入一个在词法分析中不起作用的符号到达的状态，如([{等
    DIV,          // 读入一个/
    LESS,         // 读入一个<
    GREATER,      // >
    PREP_SHARP,   // #
    ZERO,         // 0
    BASE10,       //表示十进制数的状态
    SINGLE_DOT,   //读入一个.到达的状态
    AND,          // &
    OR,           // |
    CHAR_QUOTATION,  // '
    CHAR_PREF,       // 读入'和一个字符之后到达的状态
    CHAR_END,        // 表示字符结尾的状态
    PLUS,            // 读入+到达状态
    MINUS,           // -
    EQUAL,           // =
    STR_QUOTATION,   // "
    ID,              // 表示标识符的状态
```

```
    DD_COMMA,         // 输入//之后的状态
    DS_COMMA,         // 输入/*之后的状态
    LESS_EQUAL,       // ⩽
    BIN_OPER,         // 表示二元运算符的状态
    GREATER_EQUAL,    // ⩾
    BASE8,            //表示八进制数的状态
    BASE16,           //表示十六进制数字的状态
    FLOAT_DOT,  // 表示输入了浮点数到小数点一位时到达的状态
    AAND,       // &&
    OOR,        // ||
    PPLUS,      // ++
    MMINUS,     // --
    EEQUAL,     // ==
    STR_END,    // 表示字符串结尾的状态
    DD_COMMA_END,  // 表示单行注释结束的状态
    DS_COMMA_S,    // 表示多行注释结尾输入了一个*的状态
    SELF_OPER,     // 表示自操作运算符的状态，如+=，>>=
    INT_U,         //整数结尾输入后缀U的状态
    INT_L,         // 整数结尾输入后缀L的状态
    FLOAT,         // 表示浮点数的状态
    DS_COMMA_END,  //表示多行注释结束的状态
    INT_UL,        //整数结尾输入后缀UL的状态
    INT_LL,        // 整数结尾输入后缀LL的状态
    FLOAT_E,       // 浮点数后输入e的状态
    INT_ULL,       // 整数结尾输入ULL的状态
    FLOAT_E_FLAG,  // 浮点数e后输入了一个+/-号的状态
    FL_INT,        // 浮点数e和+/-号后面跟了一个整数的状态
    FLOAT_L,       // 浮点数后加入L后缀的状态
    STR_DASH,      //正在输入的字符串中输入了一个\到达的状态
    CHAR_DASH      //正在输入的字符中输入了一个\到达的状态
};

enum class TOKEN {
    NOT_END,      // 未结束的符号
    ID,           // 表示标识符的符号
    INT,          // 整数
    FLOAT,        //浮点数
    RELOP,        // 关系比较符
    ASSIGN,       // 赋值符号
    OPER,         // 运算符
    COMMA,        // 注释
    SINGLE_MARK,  // 单个不在词法分析中产生作用的符号
    PREP,         // 预处理语句
    STR,          // 字符串
    BLANK,        // 空白符
    CHAR,         // 字符
    ERR           // 表示错误
};

const std::string TOKEN_NAME[] = {
    "NOT_END", "ID",          "INT",  "FLOAT", "RELOP", "ASSIGN", "OPER",
    "COMMA",   "SINGLE_MARK", "PREP", "STR",   "BLANK", "CHAR",   "ERR"};

using Func = std::function<bool(char)>;

Func Include(std::set<char> S) {  // 返回一个判断字符是否在集合内的function
    return [=](char c) -> bool { return S.count(c); };
}
```

```
Func Exclude(std::set<char> S) {  // 返回判断字符是否不在集合内的function
    return [=](char c) -> bool { return !S.count(c); };
}

Func IsBase8 = [](char c) -> bool {  // 返回判断字符是否是8进制串中的合法字符
    return '0' <= c && c <= '7';
};
Func IsBase16 = [](char c) -> bool {  // 返回判断字符是否是16进制串中的合法字符
    return isdigit(c) or ('A' <= c && c <= 'F');
};

Func IsBlank = [](char c) -> bool {  // 判断字符是否是空白符
    return (c == '\t' || c == ' ' || c == '\n');
};

Func Between(char l, char r) {  // 返回判断字符是否在一个指定区间的function
    return [=](char c) -> bool { return l <= c && c <= r; };
}

Func InCharSet = [&](char c) -> bool {  // 判断字符是否在支持的字符集中
    return isalpha(c) || isdigit(c) || IsBlank(c) ||
           std::set<char>({'@', '`',  '$',  // 只可能出现在string中的字符
                           '!', '"',  '#', '%', '&', '(', ')', '*', '+', ',',
                           '-', '\'', '.', '/', ':', ';', '<', '=', '>', '?',
                           '[', '\\', ']', '^', '_', '{', '|', '}', '~'})
               .count(c);
};

using Token = std::tuple<int, int, TOKEN, std::string>;

struct LexAutomata {
    struct State {                                  // 自动机状态的定义
        TOKEN type;                                 // 状态的属性
        std::vector<std::pair<Func, STATES>> go;  // 表示转移的数组
        STATES
        Go(char c) {  // 转移函数
            for (auto &[pred, s] : go)
                if (pred(c)) return s;
            return FAIL;
        }
    };
    std::vector<State> state;
    LexAutomata()  // 自动机的初始化过程
        : state(
              {// INIT
               {TOKEN::NOT_END,
                {{[](char c) -> bool { return isalpha(c) || c == '_'; }, ALPHA},
                 {Include({'!', ';', '(', ')', '[', ']', '{', '}', ',', '?',
                           ':', '~'}),
                  SINGLE_MARK},
                 {Include({'/'}), DIV},
                 {Include({'<'}), LESS},
                 {Include({'>'}), GREATER},
                 {Include({'*', '^', '%'}), BIN_OPER},
                 {Include({'#'}), PREP_SHARP},
                 {Include({'0'}), ZERO},
                 {Between('1', '9'), BASE10},
```

```
    {Include({'.'}), SINGLE_DOT},
    {Include({'&'}), AND},
    {Include({'|'}), OR},
    {Include({'+'}), PLUS},
    {Include({'-'}), MINUS},
    {Include({'='}), EQUAL},
    {Include({'"'}), STR_QUOTATION},
    {IsBlank, INIT},
    {Include({'\''}), CHAR_QUOTATION}}},
  // ALPHA
  {TOKEN::ID,
   {{[](char c) -> bool { return isalnum(c) || c == '_'; }, ID}}},
  // SINGLE_MARK
  {TOKEN::SINGLE_MARK, {}},
  // DIV
  {TOKEN::OPER,
   {{Include({'='}), SELF_OPER},
    {Include({'/'}), DD_COMMA},
    {Include({'*'}), DS_COMMA}}},
  // LESS
  {TOKEN::RELOP,
   {{Include({'='}), LESS_EQUAL}, {Include({'<'}), BIN_OPER}}},
  // GREATER
  {TOKEN::RELOP,
   {{Include({'='}), GREATER_EQUAL}, {Include({'>'}), BIN_OPER}}},
  // PREP_SHARP
  {TOKEN::PREP, {{Exclude({'\n'}), PREP_SHARP}}},
  // ZERO
  {TOKEN::INT,
   {{IsBase8, BASE8},
    {Include({'x'}), BASE16},
    {Include({'.'}), FLOAT_DOT}}},
  // BASE10
  {TOKEN::INT,
   {{isdigit, BASE10},
    {Include({'U'}), INT_U},
    {Include({'L'}), INT_L},
    {Include({'.'}), FLOAT_DOT},
    {Include({'e'}), FLOAT_E}}},
  // SINGLE_DOT
  {TOKEN::OPER, {{isdigit, FLOAT}}},
  // AND
  {TOKEN::OPER,
   {{Include({'&'}), AAND}, {Include({'='}), SELF_OPER}}},
  // OR
  {TOKEN::OPER,
   {{Include({'|'}), OOR}, {Include({'='}), SELF_OPER}}},
  // CHAR_QUOTATION
  {TOKEN::NOT_END,
   {{Include({'\\'}), CHAR_DASH},
    {Exclude({'\\', '\'', '\n'}), CHAR_PREF}}},
  // CHAR_PREF
  {TOKEN::NOT_END,
   {{Include({'\\'}), CHAR_DASH},
    {Exclude({'\\', '\'', '\n'}), CHAR_PREF},
    {Include({'\''}), CHAR_END}}},
  // CHAR_END
  {TOKEN::CHAR, {}},
```

```
                // PLUS
                {TOKEN::OPER,
                 {{Include({'+'}), PPLUS},
                  {Include({'0'}), ZERO},
                  {Between('1', '9'), BASE10},
                  {Include({'='}), SELF_OPER}}},
                // MINUS
                {TOKEN::OPER,
                 {{Include({'-'}), MMINUS},
                  {Include({'0'}), ZERO},
                  {Between('1', '9'), BASE10},
                  {Include({'='}), SELF_OPER}}},
                // EQUAL
                {TOKEN::ASSIGN, {{Include({'='}), EEQUAL}}},
                // STR_QUOTATION
                {TOKEN::NOT_END,
                 {{Include({'\\'}), STR_DASH},
                  {Exclude({'\\', '"', '\n'}), STR_QUOTATION},
                  {Include({'"'}), STR_END}}},
                // ID
                {TOKEN::ID, {{isalnum, ID}}},
                // DD_COMMA
                {TOKEN::COMMA,
                 {{Exclude({'\n'}), DD_COMMA}, {Include({'\n'}), DD_COMMA_END}}},
                // DS_COMMA,
                {TOKEN::NOT_END,
                 {{Exclude({'*'}), DS_COMMA}, {Include({'*'}), DS_COMMA_S}}},
                // LESS_EQUAL
                {TOKEN::RELOP, {}},
                // BIN_OPER
                {TOKEN::OPER, {{Include({'='}), SELF_OPER}}},
                // GREATER_EQUAL
                {TOKEN::RELOP, {}},
                // BASE8
                {TOKEN::INT,
                 {{IsBase8, BASE8},
                  {Include({'U'}), INT_U},
                  {Include({'L'}), INT_L}}},
                // BASE16
                {TOKEN::INT,
                 {{IsBase16, BASE16},
                  {Include({'U'}), INT_U},
                  {Include({'L'}), INT_L}}},
                // FLOAT_DOT
                {TOKEN::NOT_END, {{isdigit, FLOAT}}},
                // AAND
                {TOKEN::OPER, {}},
                // OOR
                {TOKEN::OPER, {}},
                // PPLUS
                {TOKEN::OPER, {}},
                // MMINUS
                {TOKEN::OPER, {}},
                // EEQUAL
                {TOKEN::RELOP, {}},
                // STR_END
                {TOKEN::STR, {}},
                // DD_COMMA_END
```

```
            {TOKEN::COMMA, {}},
            // DS_COMMA_S
            {TOKEN::NOT_END,
             {{Include({'/'}), DS_COMMA_END},
              {Exclude({'/', '*'}), DS_COMMA},
              {Include({'*'}), DS_COMMA_S}}},
            // SELF_OPER
            {TOKEN::OPER, {}},
            // INT_U
            {TOKEN::INT, {{Include({'L'}), INT_UL}}},
            // INT_L
            {TOKEN::INT, {{Include({'L'}), INT_LL}}},
            // FLOAT
            {TOKEN::FLOAT,
             {{isdigit, FLOAT},
              {Include({'e'}), FLOAT_E},
              {Include({'L'}), FLOAT_L}}},
            // DS_COMMA_END
            {TOKEN::COMMA, {}},
            // INT_UL
            {TOKEN::INT, {{Include({'L'}), INT_ULL}}},
            // INT_LL
            {TOKEN::INT, {}},
            // FLOAT_E
            {TOKEN::NOT_END,
             {{isdigit, FL_INT}, {Include({'+', '-'}), FLOAT_E_FLAG}}},
            // INT_ULL
            {TOKEN::INT, {}},
            // FLOAT_E_FLAG
            {TOKEN::NOT_END, {{isdigit, FL_INT}}},
            // FL_INT
            {TOKEN::FLOAT, {{isdigit, FL_INT}, {Include({'L'}), FLOAT_L}}},
            // FLOAT_L
            {TOKEN::FLOAT, {}},
            // STR_DASH
            {TOKEN::NOT_END, {{Exclude({'\n'}), STR_QUOTATION}}},
            // CHAR_DASH
            {TOKEN::NOT_END, {{Exclude({'\n'}), CHAR_PREF}}}}) {}

std::vector<Token> Run(
    std::ifstream &stream) {  //将流在自动机上匹配输出符号的过程
    std::string token;        //当前输入的符号前缀
    std::vector<Token> tokens;  // 输出的符号列表
    STATES s = INIT;            // 当前状态，初始设置为INIT
    std::vector<int> linec = {
        1};  // 表示每行分别已经输入了多少个字符，用于统计行号和列号
    int line = 1, col = 1;  // 当前符号的开始位置的行列
    bool eof = 0;           // 是否已经读到流的结尾
    auto Output = [&](TOKEN type) {
        auto IsNumberToken = [&](TOKEN c)
            -> bool {  // 判断是否是数字状态的函数，在下面的判断中可以少些几遍
            return c == TOKEN::INT || c == TOKEN::FLOAT;
        };

        auto Can = [&](std::pair<TOKEN, std::string> a,
                       std::pair<TOKEN, std::string> b)
            -> bool {  // 判断两个符号能否相邻的函数
            if (b.first == TOKEN::ID) return !IsNumberToken(a.first);
```

```
                if (b.first == TOKEN::PREP) return a.second == "\n";
                if (a.first == TOKEN::ERR)
                    return b.first == TOKEN::BLANK ||
                           b.first == TOKEN::SINGLE_MARK ||
                           b.first == TOKEN::OPER;
                return true;
            };
            if (tokens.empty())
                tokens.push_back({line, col, type, token});
            else {
                auto &[l0, c0, typ0, tok0] = tokens.back();
                bool primaryFail = false;
                if (IsNumberToken(type)) {  //将1和+1的形式拆成1 + 1
                    if (IsNumberToken(typ0) || typ0 == TOKEN::ID) {
                        if (token[0] == '+' || token[0] == '-') {
                            tokens.push_back(
                                {line, col, TOKEN::OPER, token.substr(0, 1)});
                            tokens.push_back(
                                {line, col + 1, type,
                                 token.substr(1, token.length() - 1)});
                        } else
                            primaryFail = true;
                    }
                }
                if (primaryFail ||
                    !Can({typ0, tok0},
                         {type, token})) {  //不能相邻的符号，合在一起组成ERR
                    std::get<2>(tokens.back()) = TOKEN::ERR;
                    std::get<3>(tokens.back()) += token;
                } else
                    tokens.push_back({line, col + 1, type, token});
            }
            line = linec.size();
            col = linec.back();
            token = "";
        };
        for (char c; (c = stream.get()) != EOF;) {
            if (!InCharSet(c)) {
                std::cerr << "Unsupported character " << c
                          << '\n';  // 不在字符集中的符号，直接忽略
            } else {
                if (c == '\n')  // 新行开始
                    linec.push_back(1);
                else
                    linec.back()++;       // 当前行多了一个字符
                auto n = state[s].Go(c);  // 计算转移到的状态
                if (s == INIT &&
                    n ==
                        INIT) {  // 只有空白分隔符能从INIT转移到INIT，此时直接输出不用处
理

                    token += c;
                    Output(TOKEN::BLANK);
                } else if (n == FAIL) {  // 失配了
                    if (state[s].type ==
                        TOKEN::
                            NOT_END) {  // 上一个状态不是终止态，则认为发生了错误
                        token += c;
                        Output(TOKEN::ERR);  //报错
```

```
                    } else {
                        stream.unget();  // 回退一个字符
                        if (c == '\n')
                            linec.pop_back();
                        else
                            linec.back()--;
                        Output(state[s].type);  // 将已经结束的符号输出
                    }
                    s = INIT;
                } else {  //没有发生什么问题，平平无奇的转移
                    s = n;
                    token += c;
                }
            }
        }
        if (token != "")
            Output(state[s].type == TOKEN::NOT_END ? TOKEN::ERR
                                                   : state[s].type);
        return tokens;
    }
} lam;

namespace Formatter {
static const std::string COLOR[] = {
    //在Linux Shell中控制颜色的字符串
    "\e[0;31m"
    "\e[0;32m",
    "\e[1;33m",
    "\e[1;34m",
    "\e[1;35m",
};
#define REVERT_COLOR "\e[7m"
#define RESET_COLOR "\033[0m"
#define FADE_COLOR "\033[2m"
void Code(std::vector<Token> output) {
    int lino = 0;
    for (auto [l, c, typ, tok] : output) {
        auto GColor = []() -> std::string {
            static int cc;
            return COLOR[cc++ % 4];
        };
        auto color = GColor();  //循环使用4种颜色输出符号，便于区分辨认
        if (typ == TOKEN::ERR)
            color += REVERT_COLOR;  //对于错误的符号，用反色来强调
        if (lino == 0) tok = '\n' + tok;
        for (char c : tok) {
            if (c == '\n')
                std::cerr << RESET_COLOR << c << FADE_COLOR << std::setw(3)
                          << std::setfill(' ') << ++lino << '\t';  // 输出行号
            else
                std::cerr << RESET_COLOR + color << c;
        }
        std::cerr << RESET_COLOR;
    }
    std::cerr << "\n\n" << lino << " lines in total.\n";
}

void TokenList(std::vector<Token> output) {
```

```
    for (auto &[l, c, typ, tok] : output)
        if (typ != TOKEN::BLANK && typ != TOKEN::COMMA && typ != TOKEN::PREP) {
            std::cerr << std::setw(30) << std::setfill(' ') << std::left
                      << std::string(RESET_COLOR FADE_COLOR "line " +
                                     std::to_string(l) + ", column " +
                                     std::to_string(c) + ": ");
            std::cerr << RESET_COLOR << COLOR[0] << "<" << COLOR[1];
            if (typ == TOKEN::ERR) std::cerr << REVERT_COLOR;
            std::cerr << TOKEN_NAME[(int)typ];
            std::cerr << RESET_COLOR << COLOR[0] << "," << COLOR[2];
            for (char c : tok)
                if (c == '\n')
                    std::cerr << "\\n";
                else if (c == '\t')
                    std::cerr << "\\t";
                else
                    std::cerr << c;
            std::cerr << COLOR[0] << ">\n";
        }
    std::cerr << RESET_COLOR;
}
}  // namespace Formatter

int main(int argc, const char **argv) {
    std::ifstream src(argv[1]);
    auto tokens = lam.Run(src);
    src.close();
    std::cerr << "Colored Source Code:\n";
    Formatter::Code(tokens);
    std::cerr << "\n\nToken List:\n";
    Formatter::TokenList(tokens);
    std::cerr << RESET_COLOR "\n\nToken Counts:\n";
    std::map<std::string, int> tcount;
    int ccount = 0;
    for (auto &[l, c, typ, tok] : tokens) {
        tcount[TOKEN_NAME[(int)typ]]++;
        ccount += tok.length();
    }
    for (auto &[t, c] : tcount)
        std::cerr << "Number of " << std::setw(15) << std::right << t << '\t'
                  << c << '\n';

    std::cerr << RESET_COLOR "\n\nChar Counts: " << ccount << '\n';
    return 0;
}
```

Lex

```
%{

enum {
    END,
    ID,           // 表示标识符的符号
    INT,          // 整数
    FLOAT,        //浮点数
    RELOP,        // 关系比较符
    ASSIGN,       // 赋值符号
```

```
    OPER,          // 运算符
    COMMA,         // 注释
    SINGLE_MARK,   // 单个不在词法分析中产生作用的符号
    PREP,          // 预处理语句
    STR,           // 字符串
    BLANK,         // 空白符
    CHAR,          // 字符
    ERR            // 表示错误
};
const char TOKEN_NAME[][20]={
    "","ID", "INT",  "FLOAT", "RELOP", "ASSIGN", "OPER",
    "COMMA",   "SINGLE_MARK", "PREP", "STR",   "BLANK", "CHAR",   "ERR"
};

%}

%%

[ \t\n]
          {  }
(_|[A-Za-z])(_|[A-Za-z0-9])*
           { return ID; }
[+|-]?((([1-9])([0-9])*)|(0x[0-9A-Z]+)|(0[0-7]*))(U|L|UL|LL|ULL)?
          { return INT; }
[+|-]?((([1-9][0-9]*)\.[0-9]+(e[+|-]?[1-9][0-9]*)?)|[1-9][0-9]*e[+|-]?[1-9][0-
9]*)(L)?      { return FLOAT; }
"<"
          { return RELOP; }
"⩽"
           { return RELOP; }
">"
          { return RELOP; }
"⩾"
           { return RELOP; }
"⩵"
           { return RELOP; }
"="
          { return ASSIGN; }
"+"
          { return OPER; }
"-"
          { return OPER; }
"*"
          { return OPER; }
"/"
          { return OPER; }
">>"
           { return OPER; }
"<<"
           { return OPER; }
"&"
          { return OPER; }
"|"
          { return OPER; }
"^"
          { return OPER; }
(\/\/[^\n]*\n)|(\/\*[^*]*\*+([^/*][^*]*\*+)*\/)
         { return COMMA; }
```

```
"("
          { return SINGLE_MARK; }
")"
          { return SINGLE_MARK; }
"["
          { return SINGLE_MARK; }
"]"
          { return SINGLE_MARK; }
"{"
          { return SINGLE_MARK; }
"}"
          { return SINGLE_MARK; }
"."
          { return SINGLE_MARK; }
"!"
          { return SINGLE_MARK; }
"~"
          { return SINGLE_MARK; }
";"
          { return SINGLE_MARK; }
":"
          { return SINGLE_MARK; }
"?"
          { return SINGLE_MARK; }
","
          { return SINGLE_MARK; }
#[^\n]*\n
          { return PREP; }
\"([^\\^\n]|(\\[^\n]))*\"
          { return STR; }
'([^\\^\n]|(\\[^\n]))+'
          { return CHAR; }
.
          { return ERR; }
%%

int main (int argc, char ** argv){
    yyin=fopen(argv[1],"r");
#define RESET_COLOR "\033[0m"
#define FADE_COLOR "\033[2m"
    for (char c; (c = yylex()) != 0; )
        fprintf(yyout,FADE_COLOR"
<"RESET_COLOR"%s"FADE_COLOR","RESET_COLOR"%s"FADE_COLOR">\n",TOKEN_NAME[c],yytext
);
    fclose(yyin);
    return 0;
}
```